The Session Abstract Machine

Luís Caires

NOVALINCS
LABORATORY FOR COMPUTER SCIENCE AND INFORMATICS

NOVA
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

→ TÉCNICO LISBOA

inesc id lisboa

July 1st 2023

IFIP WG2.2. Meeting Bologna, September 2023

CLASS code (arithmetic server)

```
type tmenu {
    offer of {
        |#Neg:  recv ~lint; send lint; close
        |#Add:  recv ~lint; recv ~lint; send lint; close
    }
};;


proc server(s:!tmenu) {
    !s(c);
    case c of {
        |#Neg:  recv c(n);
                send c(v. let v -n);
                close c
        |#Add:  recv c(n1);
                recv c(n2);
                send c(v. let v n1 + n2);
                close c
    }
};;

proc client1( ; s:~tmenu){
    call s(c);
    #Neg c;
    send c (v:lint. let v 2);
    recv c(m);
    wait c;
    println("CLIENT1 GOT NEG 2 = " + m);
    ()
};;
```

```
proc client1( ; s:~tmenu){
    call s(c);
    #Neg c;
    send c (v:lint. let v 2);
    recv c(m);
    wait c;
    println("CLIENT1 GOT NEG 2 = " + m);
    ()
};;


proc client2( ; s:~tmenu){
    call s(c);
    #Add c;
    send c(2);
    send c(3);
    recv c(m);
    wait c;
    println("CLIENT2 GOT ADD 2 3 =  " + m);
    ()
};;

proc system(){
    cut{
        server(s)
        |s:?~tmenu|
        par{
            client1(;s)  || client2(;s)
        }
    }
};;
```

CLASS code (hoare-like monitor)

```
type corec CIncI {
        coaffine IncI }
and IncI {
        offer of {
        | #Inc: CIncI
        | #End: wait
        }
};;

type corec CDecI { coaffine DecI }
and DecI {
        offer of {
        | #Dec: coaffine recv ContDec; wait
        | #Share: recv CDecI; CDecI
        | #End: wait
        }
} and ContDec { coaffine send CDecI; close
};;

type rec Rep {
        send !lint;
        affine  send WaitQ(DecI); WaitQ(IncI)
} and WaitQ(I) {
  affine choice of {
        | #Next: NodeQ(I)
        | #Null: close
        }
} and ContW(I) {
        affine recv ~affine Rep;
           send affine                 ine I; wait
} and NodeQ(I) {
        state send ContW(I), WaitQ(I)
};;
```

```
proc rec decloop(dv:DecI, m:usage ~Rep)
{
        case dv of {
        |#Share:   recv dv (dvn);
                   share m { decloop(dvn,m) || decloop(dv,m) }
        |#Dec:
                recv dv(acc);
                wait dv;
                take m(n);
                recv n(val);
                if (val>0) {
                        put m(v. affine v; send v (val-1); fwd n v);
                        println "Dec "+val;
                        send acc (coo. decloop(coo,m));
                        close acc
                        } {
                        recv n(qd);
                        println "Dec put on Wait "+val;
                        letc | decc: ~ContW(DecI) |
                                awaitNZ(m, val, qd, n, decc)
                        in
                                affine decc;
                                recv decc(s0);
                                recv s0(v);
                                send decc(s.affine s; send s (v-1); fwd s0 s);
                                println "Dec";
                                fwd acc decc
                        }
        |#End: wait dv; release m
        }
};;
```

Propositions-as-Types

- *Bridge* between Logic, Programming Languages, and Computation.
- Programs are proofs in a logic, according to a Curry-Howard correspondence
    - program as a typed semantically well-behaved object (a function or a process)
    - proof simplification as computation ➡
        - preservation, progress, confluence
    - computation as cut-elimination ➡
        - logical relations semantics, termination
        - equational reasoning about observational equivalence

Propositions-as-Types for Concurrency

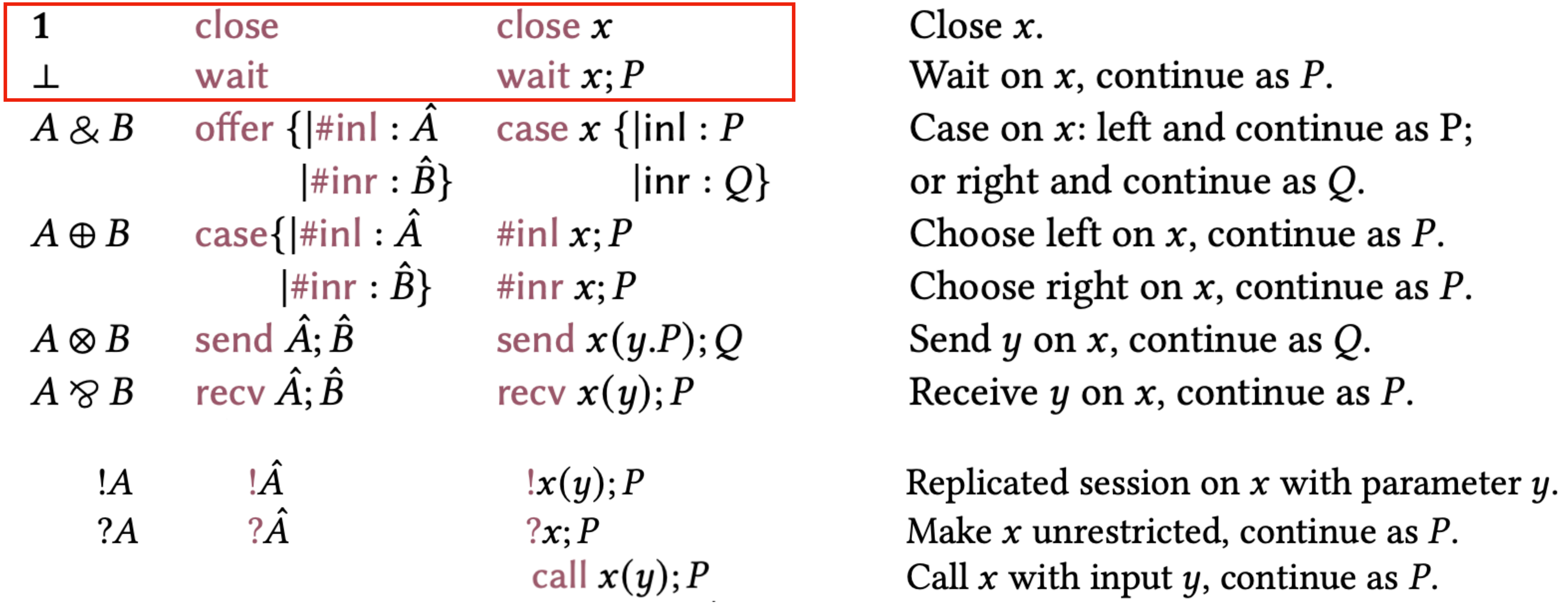
- Linear Logic and Session Types [CairesPfenning10,Wadler12]
- modular extensions (logically inspired connectives “automatically” socialize)
    - **ho-functions / polymorphism / recursion** [ToninhoCairesPfenning13]
    - **dependent types** (assertions, certificates, …) [ToninhoCairesPfenning11]
    - **effects** (discardable resources, exceptions, non-determinism) [CairesPerez17]
    - **shared state** [BalzerPfenning19,RochaCaires21]
- Towards shared state programs that can prove themselves
    - **CLASS** (RochaCaires23)

Propositions-as-Types for Concurrency

- *Bridge* between Logic, Programming Languages, and Computation.
- Bringing together process algebra and classical computation theory
  - Connecting session types to the trunk of "classical" type theory
    - typed λ-calculus: **sequential** ho computation with **pure values**
    - typed session calculus: **concurrent** ho computation with **linear resources**
    - latter subsumes former, via exponentials and sharing constructs
  - foundational infrastructure for safe concurrent programming (cf. Rust, Move)
- **THIS TALK**: Abstract Machine Semantics (**new**)

A Session Programming Language from Linear Logic

Process expressions (basic)



Logical Type	PL type	PL construct	
$\mathbf{1}$	close	close x	Close x.
$\perp$	wait	wait $x; P$	Wait on x, continue as P.
$A \mathbin{\&} B$	offer {|#inl : $\hat{A}$ |#inr : $\hat{B}$}	case x {|inl : P |inr : Q}	Case on x: left and continue as P; or right and continue as Q.
$A \oplus B$	case{|#inl : $\hat{A}$ |#inr : $\hat{B}$}	#inl $x; P$ #inr $x; P$	Choose left on x, continue as P. Choose right on x, continue as P.
$A \otimes B$	send $\hat{A}; \hat{B}$	send $x(y.P); Q$	Send y on x, continue as Q.
$A \parr B$	recv $\hat{A}; \hat{B}$	recv $x(y); P$	Receive y on x, continue as P.
$!A$	$!\hat{A}$	$!x(y); P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x; P$	Make x unrestricted, continue as P.
		call $x(y); P$	Call x with input y, continue as P.

Process expressions (basic)

Logical Type	PL type	PL construct	
1	close	close x	Close x.
$\bot$	wait	wait $x; P$	Wait on x, continue as P.
$A \& B$	offer $\{\vert\#\mathsf{inl} : \hat{A}$ $\vert\#\mathsf{inr} : \hat{B}\}$	case x $\{\vert\mathsf{inl} : P$ $\vert\mathsf{inr} : Q\}$	Case on x: left and continue as P; or right and continue as Q.
$A \oplus B$	case$\{\vert\#\mathsf{inl} : \hat{A}$	#inl $x; P$	Choose left on x, continue as P.
	$\vert\#\mathsf{inr} : \hat{B}\}$	#inr $x; P$	Choose right on x, continue as P.
$A \otimes B$	send $\hat{A}; \hat{B}$	send $x(y.P); Q$	Send y on x, continue as Q.
$A \parr B$	recv $\hat{A}; \hat{B}$	recv $x(y); P$	Receive y on x, continue as P.
$!A$	$!\hat{A}$	$!x(y); P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x; P$	Make x unrestricted, continue as P.
		call $x(y); P$	Call x with input y, continue as P.

Process expressions (basic)

Logical Type	**PL type**	**PL construct**	
$\mathbf{1}$	close	close x	Close x.
$\bot$	wait	wait $x; P$	Wait on x, continue as P.
$A \mathbin{\&} B$	offer {|#inl : $\hat{A}$	case x {|inl : P	Case on x: left and continue as P;
	|#inr : $\hat{B}$}	|inr : Q}	or right and continue as Q.
$A \oplus B$	case{|#inl : $\hat{A}$	#inl $x; P$	Choose left on x, continue as P.
	|#inr : $\hat{B}$}	#inr $x; P$	Choose right on x, continue as P.
$A \otimes B$	send $\hat{A}; \hat{B}$	send $x(y.P); Q$	Send y on x, continue as Q.
$A \parr B$	recv $\hat{A}; \hat{B}$	recv $x(y); P$	Receive y on x, continue as P.
$!A$	$!\hat{A}$	$!x(y); P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x; P$	Make x unrestricted, continue as P.
		call $x(y); P$	Call x with input y, continue as P.

Process expressions (intro / slim)

Logical Type	PL type	PL construct	
$\mathbf{1}$	close	close x	Close x.
$\perp$	wait	wait $x; P$	Wait on x, continue as P.
$A \mathbin{\&} B$	offer $\{$|#inl : $\hat{A}$ |#inr : $\hat{B}\}$	case x $\{$|inl : P |inr : $Q\}$	Case on x: left and continue as P; or right and continue as Q.
$A \oplus B$	case$\{$|#inl : $\hat{A}$ |#inr : $\hat{B}\}$	#inl $x; P$	Choose left on x, continue as P.
		#inr $x; P$	Choose right on x, continue as P.
$A \otimes B$	send $\hat{A}; \hat{B}$	send $x(y.P); Q$	Send y on x, continue as Q.
$A \parr B$	recv $\hat{A}; \hat{B}$	recv $x(y); P$	Receive y on x, continue as P.
$!A$	$!\hat{A}$	$!x(y); P$	Replicated session on x with parameter y.
$?A$	$?\hat{A}$	$?x; P$	Make x unrestricted, continue as P.
		call $x(y); P$	Call x with input y, continue as P.

Process expressions (composition)

$\mathbf{fwd}\ x\ y$

$\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}$

$\mathbf{par}\ \{P \mid\mid Q\}$

$$\overline{\mathbf{1}} \triangleq \bot$$
$$\overline{A \otimes B} \triangleq \overline{A} \parr \overline{B}$$
$$\overline{A \oplus B} \triangleq \overline{A}\ \&\ \overline{B}$$
$$\overline{!A} \triangleq ?\overline{B}$$

Duality

Congruence / Reduction Semantics

$$\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}\ \equiv \mathbf{cut}\ \{Q\ \ |x : \overline{A}|\ \ P\}$$

$$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{cut}\ \{Q\ \ |y|\ \ R\}\} \equiv \mathbf{cut}\ \{\mathbf{cut}\ \{P\ \ |x|\ \ Q\}\ \ |y|\ \ R\}$$

$$\mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ \ |y|\ \ P\} \longrightarrow \{x/y\}P$$

$$\mathbf{cut}\ \{\mathbf{close}\ x\ \ |x|\ \ \mathbf{wait}\ x; P\} \longrightarrow P$$

$$\mathbf{cut}\ \{\mathbf{send}\ x(y.P); Q\ \ |x|\ \ \mathbf{recv}\ x(z); R\} \longrightarrow \mathbf{cut}\ \{Q\ \ |x|\ \ \mathbf{cut}\ \{P\ \ |y|\ \ \{y/z\}R\}\}$$

$$\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\}\ \ |x|\ \ x.\mathbf{inl}; R\} \longrightarrow \mathbf{cut}\ \{P\ \ |x|\ \ R\}$$

$$\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\}\ \ |x|\ \ x.\mathbf{inr}; R\} \longrightarrow \mathbf{cut}\ \{Q\ \ |x|\ \ R\}$$

Linear Logic as a Session Programming Language

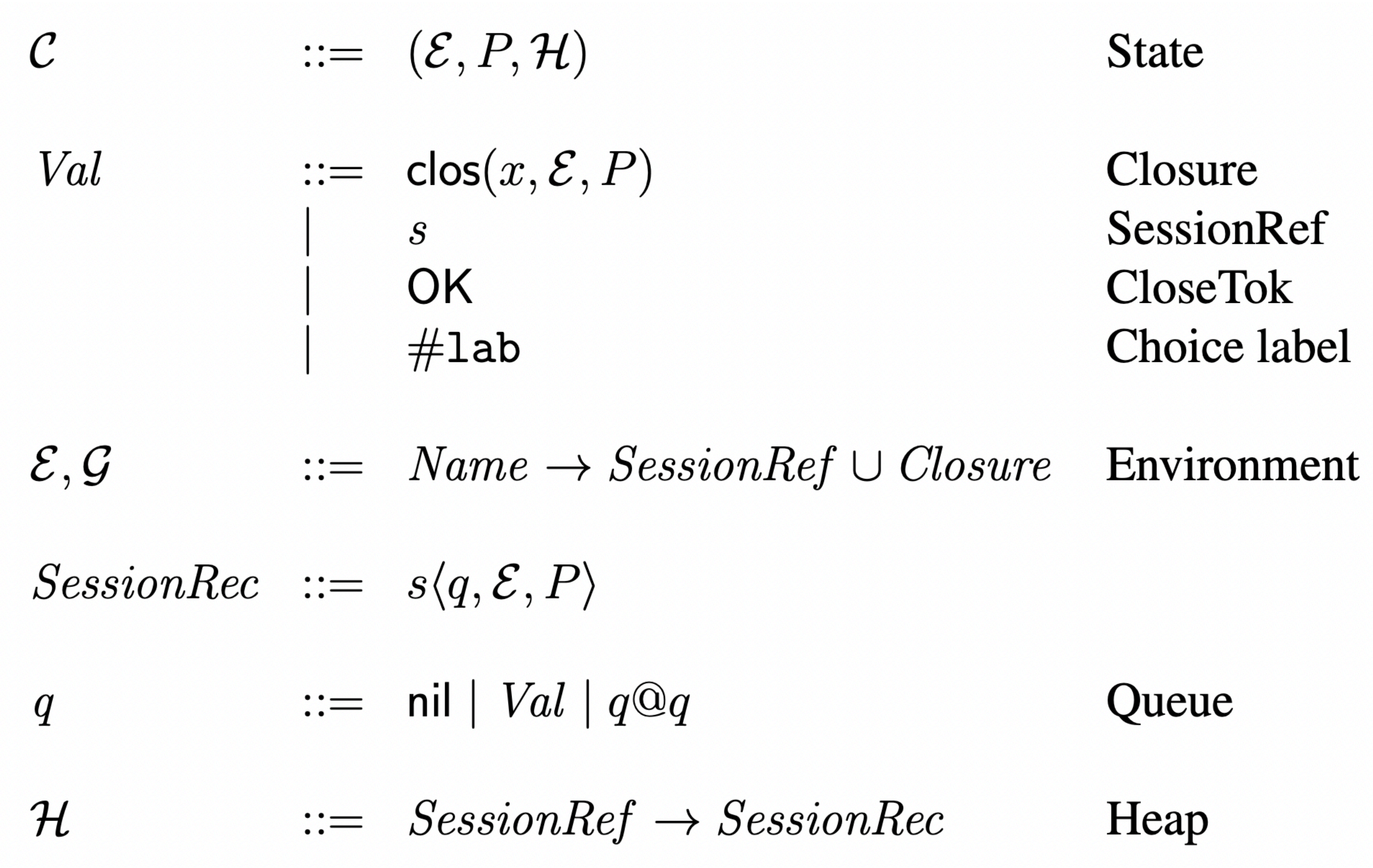
- Computational Interpretation of Linear Logic: congruence $\equiv$, reduction $\rightarrow$.
- Type Preservation: If $P \vdash \Delta; \Gamma$ and $P \rightarrow Q$, then $Q \vdash \Delta; \Gamma$.
- Deadlock-Freedom: Let $P \vdash \emptyset; \emptyset$ be a live process. Then, P reduces.
- Confluence (with sums): If $R \overset{*}{\leftarrow} P \overset{*}{\rightarrow} Q$, then exists S s.t. $R \equiv \overset{*}{\rightarrow} S \overset{*}{\leftarrow} \equiv Q$.
- Normalisation: If $P \vdash \Delta; \Gamma$, then exists a normal form Q s.t. $P \approx Q$.
- Strong Normalisation: If $P \vdash \emptyset; \emptyset$ then P is strongly normalising.

The Session Abstract Machine

Towards the Session Abstract Machine

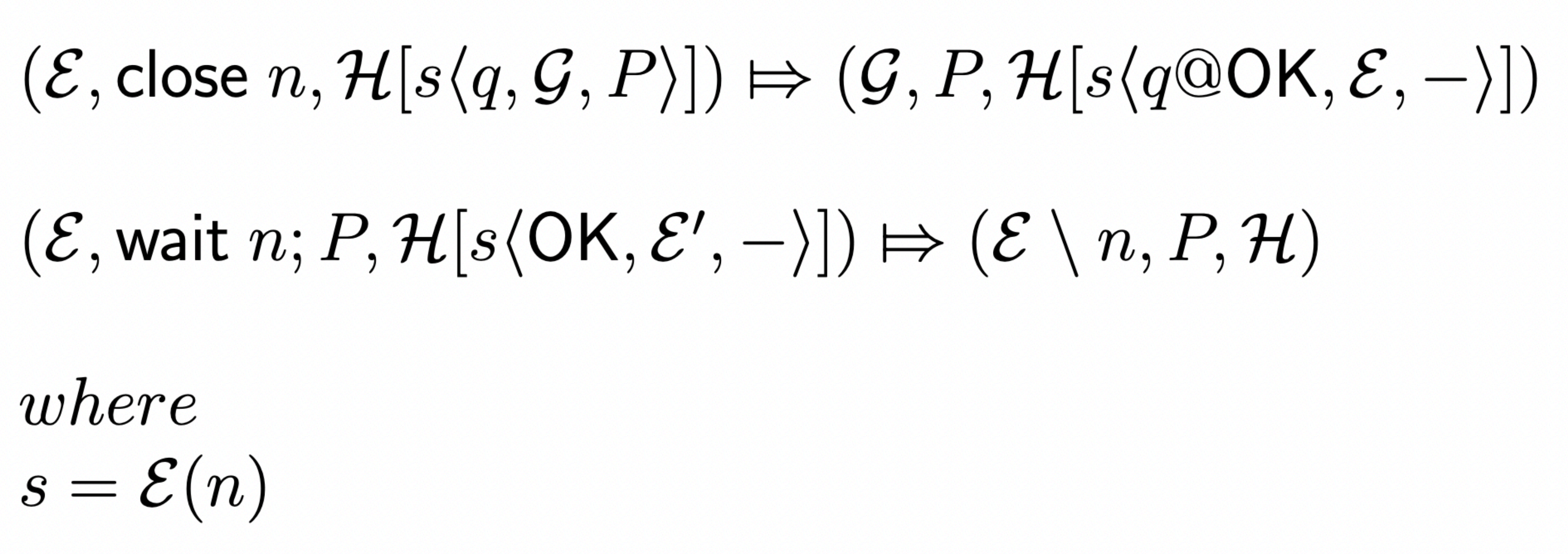
- Cf, the SECD [Landin64], LAM [Lafont88,Abramsky93], CAM [Curien86], KM[07]
- Key insights (coming from playing around with many concurrent CLASS programs)
  - Execute session terms **sequentially** whenever possible (except for Mix and Share)
  - Replace busy waiting message passing by single threaded co-routining
  - Heap-allocated mutable session object frames
  - exploit connective polarities (cf. focusing)
    - schedule positive constructs (send, select, close, bang) first
    - Split interactions in write / read moves (cf. game-semantics and SACDS)
  - respect std synchronous semantics (even if relying on buffered communication)

The Session Abstract Machine



$\mathcal{C}$	$::=$	$(\mathcal{E}, P, \mathcal{H})$	State
Val	$::=$	$\mathsf{clos}(x, \mathcal{E}, P)$	Closure
	$\mid$	s	SessionRef
	$\mid$	OK	CloseTok
	$\mid$	$\#\texttt{lab}$	Choice label
$\mathcal{E}, \mathcal{G}$	$::=$	$Name \rightarrow SessionRef \cup Closure$	Environment
$SessionRec$	$::=$	$s\langle q, \mathcal{E}, P\rangle$	
q	$::=$	$\mathsf{nil} \mid Val \mid q@q$	Queue
$\mathcal{H}$	$::=$	$SessionRef \rightarrow SessionRec$	Heap

SAM (close / wait)



$$(\mathcal{E}, \mathsf{close}\ n, \mathcal{H}[s\langle q, \mathcal{G}, P\rangle]) \Mapsto (\mathcal{G}, P, \mathcal{H}[s\langle q@\mathsf{OK}, \mathcal{E}, -\rangle])$$

$$(\mathcal{E}, \mathsf{wait}\ n; P, \mathcal{H}[s\langle \mathsf{OK}, \mathcal{E}', -\rangle]) \Mapsto (\mathcal{E} \setminus n, P, \mathcal{H})$$

$where$

$s = \mathcal{E}(n)$

SAM (cut)

$$(\mathcal{E}, \mathsf{cut}\ \{\ P\ |x:T|\ Q\}, \mathcal{H}) \Rrightarrow (\mathcal{G}, Q, \mathcal{H}[\ s\langle \mathsf{nil}, \mathcal{G}, P\rangle])$$

T positive

$$(\mathcal{E}, \mathsf{cut}\ \{\ P\ |x:T|\ Q\}, \mathcal{H}) \Rrightarrow (\mathcal{G}, P, \mathcal{H}[\ s\langle \mathsf{nil}, \mathcal{G}, Q\rangle])$$

T negative

$where$

$s = \mathsf{new}(\mathcal{H})$

$\mathcal{G} = \mathcal{E}\{s/x\}$

SAM (send / receive)

$$
\begin{aligned}
&(\mathcal{E}, \mathsf{send}\ n(y.Q); P, \mathcal{H}[s\langle q, \mathcal{E}', R\rangle]) \Mapsto \\
&\qquad (\mathcal{E}, P, \mathcal{H}[s\langle q@\mathsf{clos}(y, Q, \mathcal{E}), \mathcal{E}', R\rangle])
\end{aligned}
$$

n:send A;B (B positive)

$$
\begin{aligned}
&(\mathcal{E}, \mathsf{send}\ n(y.Q); P, \mathcal{H}[s\langle q, \mathcal{E}', R\rangle]) \Mapsto \\
&\qquad (\mathcal{E}', R, \mathcal{H}[s\langle q@\mathsf{clos}(y, Q, \mathcal{E}), \mathcal{E}, P\rangle])
\end{aligned}
$$

n:send A;B (B negative)

$where$

$s = \mathcal{E}(n)$

SAM (send / receive)

$$\begin{aligned}&(\mathcal{E}, \textsf{recv}\ n(x); P, \mathcal{H}[s\langle \textsf{clos}(y, Q, \mathcal{F}) :: q, \mathcal{G}, R\rangle]) \Mapsto \\ &\qquad (\mathcal{E}_r, P, \mathcal{H}[s\langle q, \mathcal{G}, R\rangle][k\langle \textsf{nil}, \mathcal{E}_s, Q\rangle]\end{aligned}$$

n:recv A;B (A positive)

$$\begin{aligned}&(\mathcal{E}, \textsf{recv}\ n(x); P, \mathcal{H}[s\langle \textsf{clos}(y, Q, \mathcal{F}) :: q, \mathcal{G}, R\rangle]) \Mapsto \\ &\qquad (\mathcal{E}_s, Q, \mathcal{H}[s\langle q, \mathcal{G}, R\rangle][k\langle \textsf{nil}, \mathcal{E}_r, P\rangle]\end{aligned}$$

n:recv A;B (A negative)

$$\begin{aligned}&where\\ &s = \mathcal{E}(n)\\ &\mathcal{E}_s = \mathcal{F}\{k/y\}\\ &\mathcal{E}_r = \mathcal{E}\{k/x\}\end{aligned}$$

SAM (select / case)

$$(\mathcal{E}, \texttt{\#l}\ n; P, \mathcal{H}[s\langle q, \mathcal{E}', R\rangle]) \Mapsto (\mathcal{E}, P, \mathcal{H}[s\langle q@\texttt{\#l}, \mathcal{E}', R\rangle])$$

$$\textit{where}$$

$$s = \mathcal{E}(n)$$

n:select {#l: B, … } (B positive)

$$(\mathcal{E}, \texttt{\#l}\ n; P, \mathcal{H}[s\langle q, \mathcal{E}', R\rangle]) \Mapsto (\mathcal{E}', R, \mathcal{H}[s\langle q@\texttt{\#l}, \mathcal{E}, P\rangle])$$

$$\textit{where}$$

$$s = \mathcal{E}(n)$$

n:select {#l: B, … } (B negative)

SAM (select / case)

$$
\begin{array}{l}
(\mathcal{E}, \textsf{case}\ n\ \textsf{of}\ \{\ldots, \#\texttt{l}_k{:}P_k, \ldots\}, \mathcal{H}[s\langle \#\texttt{l}_k :: q, \mathcal{G}, R\rangle]) \Mapsto \\
\qquad (\mathcal{E}_r, P_k, \mathcal{H}[s\langle q, \mathcal{G}, R\rangle] \\
\mathit{where} \\
s = \mathcal{E}(n)
\end{array}
$$

SAM (fwd)

$$
\begin{aligned}
&(\mathcal{E}, \textsf{fwd}\ m\ n, \mathcal{H}[w\langle \textsf{q}_w, \mathcal{E}_w, P\rangle][r\langle \textsf{q}_r, \mathcal{E}_r, Q\rangle]) \\
&\qquad \Mapsto (\mathcal{E}_w, P, \mathcal{H}[\ w\langle \textsf{q}_w @ \textsf{q}_r, \mathcal{E}_r\{r \to w\}, Q\rangle])
\end{aligned}
$$

n:T negative

$where$

$w = \mathcal{E}(m); r = \mathcal{E}(n)$

SAM (! / ? / call)

$$(\mathcal{E}, !n(x); P, \mathcal{H}[s\langle q, \mathcal{G}, R\rangle]) \Mapsto (\mathcal{G}, R, \mathcal{H}[s\langle q@\mathsf{clos}(x, \mathcal{E}, P), \mathcal{E}, -\rangle]$$

$where$

$s = \mathcal{E}(n)$

SAM (! / ? / call)

$$(\mathcal{E}, ?n; P, \mathcal{H}[s\langle \mathsf{clos}(x, \mathcal{G}, R), \mathcal{F}, -\rangle]) \Mapsto (\mathcal{E}', P, \mathcal{H})$$

$where$
$s = \mathcal{E}(n)$
$\mathcal{E}' = \mathcal{E}\{\mathsf{clos}(x, \mathcal{G}, R)/n\}$

SAM (! / ? / call)

$$(\mathcal{E}, \mathsf{call}\ n(y); P, \mathcal{H}) \Mapsto (\mathcal{E}', P, \mathcal{H}[s\langle \mathsf{nil}, \mathcal{G}', R\rangle])$$

n : ?A (A positive)

$$\begin{aligned}
&where\\
&s = \mathsf{new}(\mathcal{H})\\
&\mathcal{E}(n) = \mathsf{clos}(x, \mathcal{G}, R)\\
&\mathcal{E}' = \mathcal{E}\{s/y\}\\
&\mathcal{G}' = \mathcal{G}\{s/y\}
\end{aligned}$$

Correctness

Let $P \vdash \emptyset; \emptyset$.

- Completeness

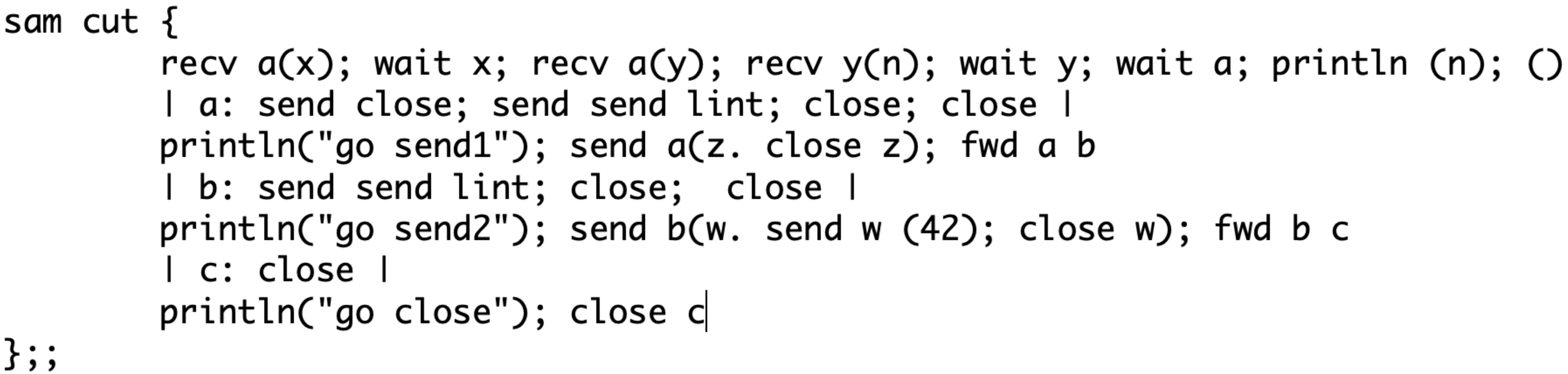
If $live(P)$ then there is $\mathcal{C}$ such that $Comp(P) \overset{*}{\Mapsto}_e \Mapsto_d \mathcal{C}$.

- Soundness

If $Comp(P) \overset{*}{\Mapsto}_e \Mapsto_d \mathcal{C}$ then
there is Q such that $P \to Q$ and $Comp(Q) \overset{*}{\Mapsto}_e \mathcal{C}$.

- NB. *Comp*(*P*) essentially decomposes cuts and fwd to expose a first action pref

SAM (example)



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

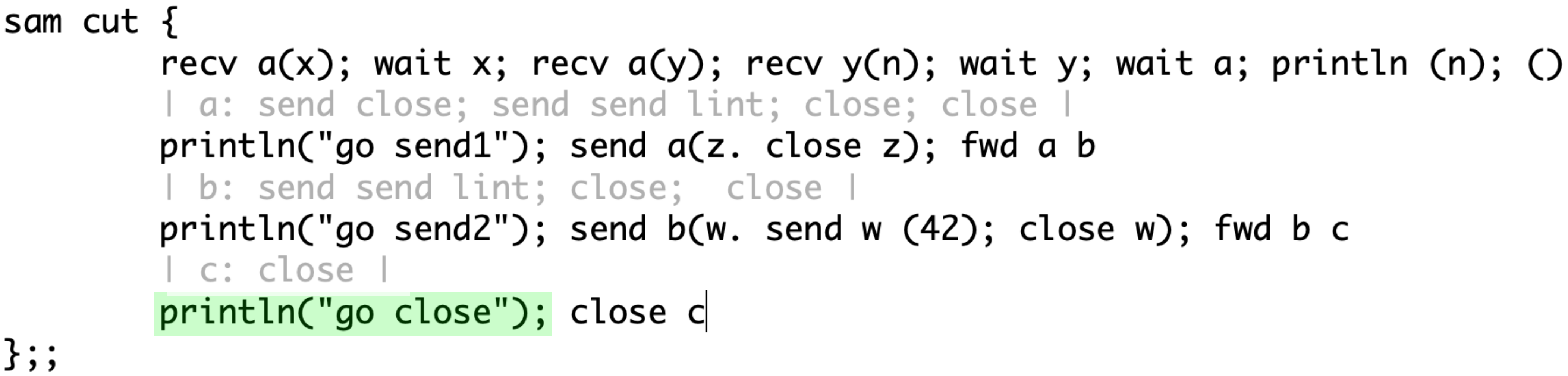
SAM (example)

a	b	c
nil	nil	nil

```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

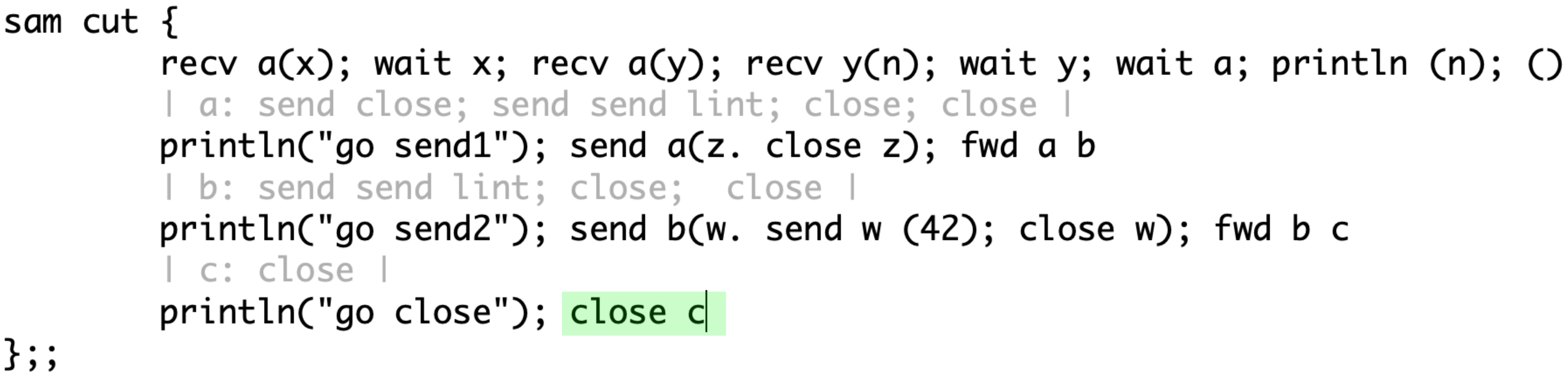
a	b	c
nil	nil	nil



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

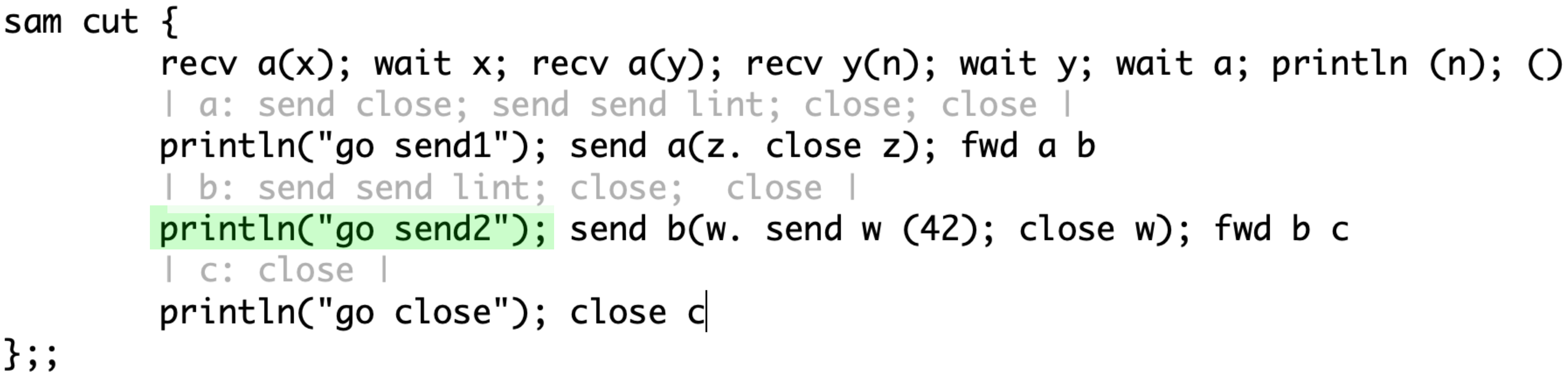
a	b	c
nil	nil	OK



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

a	b	c
nil	nil	OK



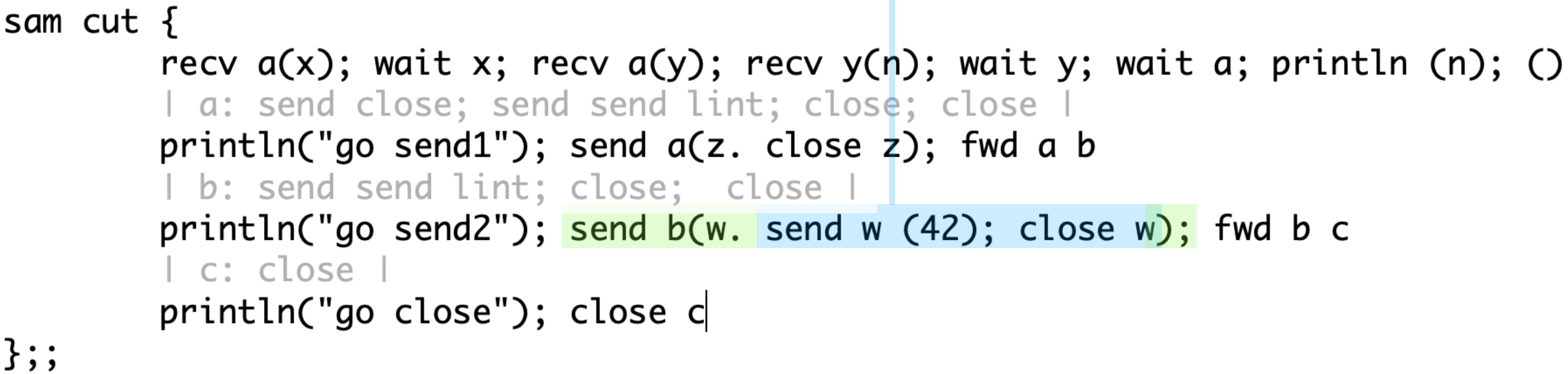
```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

a nil

b c(w,$\mathcal{E}_w$,)

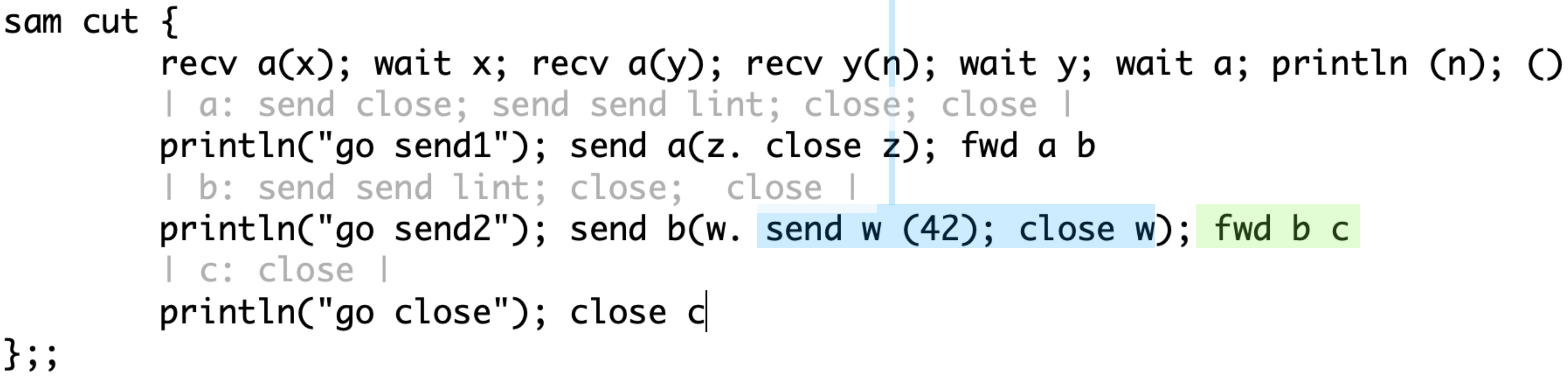
c OK



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

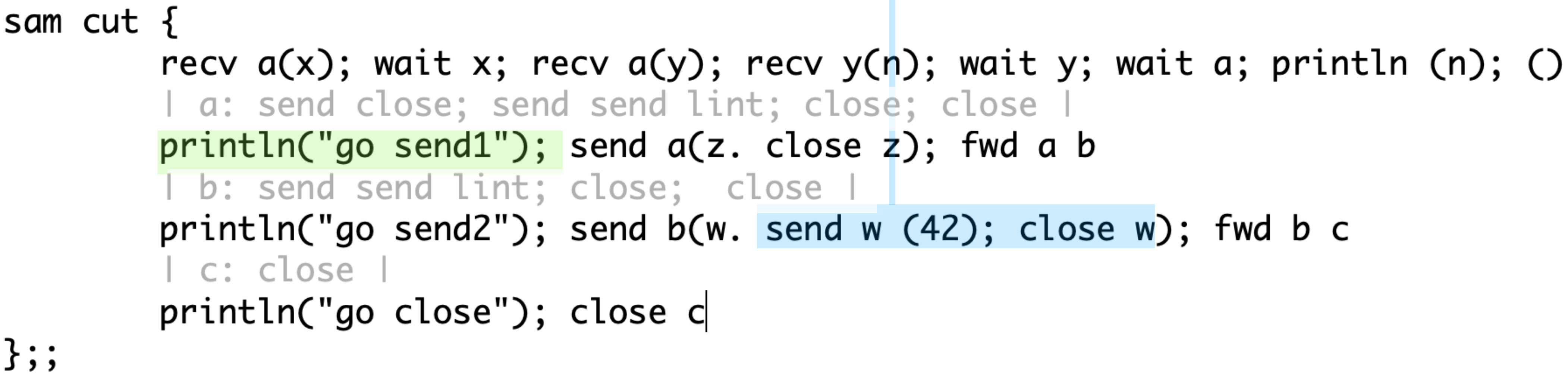
a	b
nil	$c(w, \mathcal{E}_w, \ )$ :: OK



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

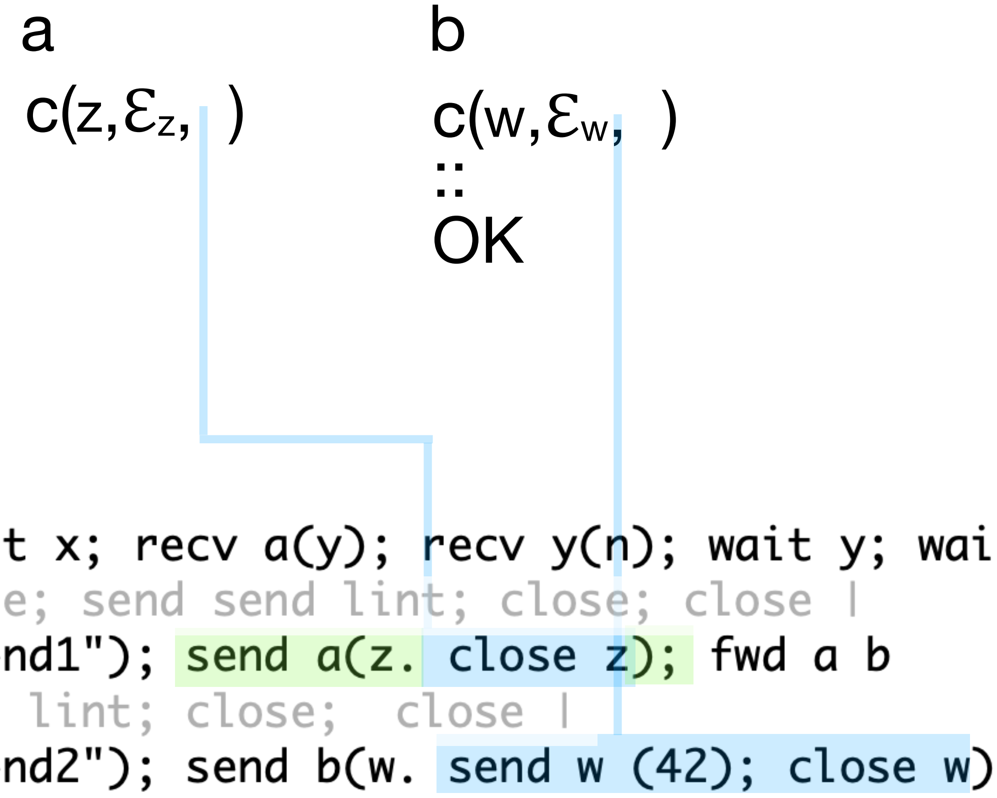
SAM (example)

a	b
nil	$c(w, \mathcal{E}_w, \ )$ :: OK



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)



a

c(z,$\mathcal{E}_z$,)

b

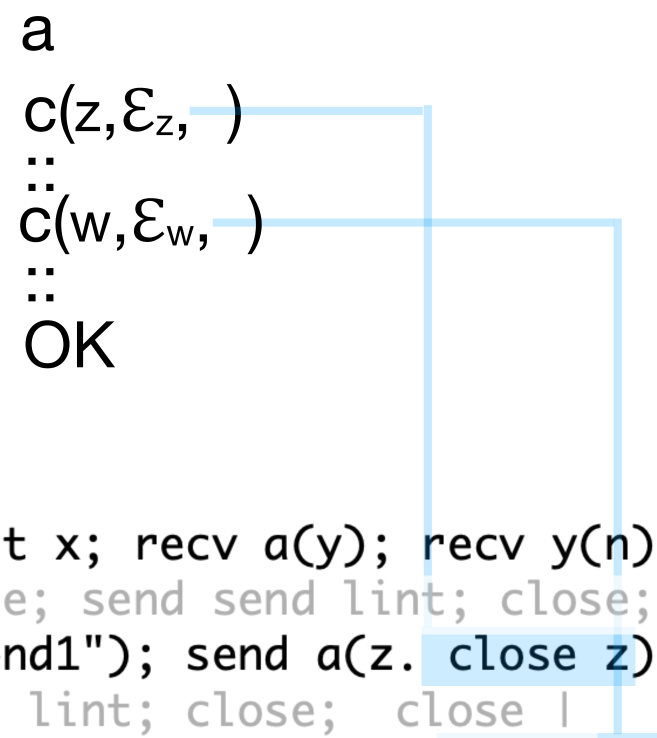
c(w,$\mathcal{E}_w$,)

::

OK

```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)



a

$c(z, \mathcal{E}_z, \ )$

::

$c(w, \mathcal{E}_w, \ )$

::

OK

```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

s	a
nil	c(w,$\mathcal{E}_w$,) :: OK

```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

s	a
OK	$c(w,\mathcal{E}_w,\ )$ :: OK

```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

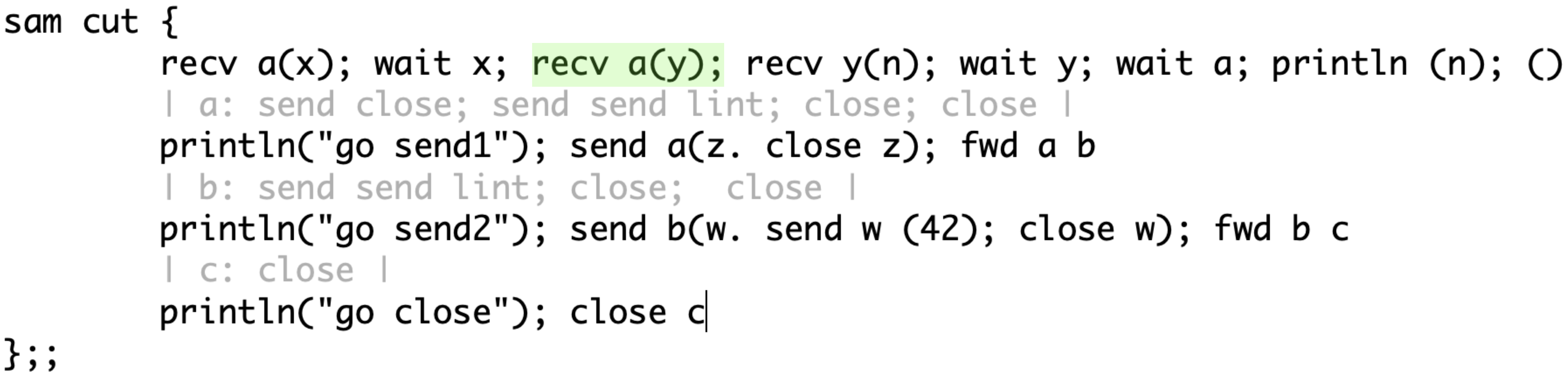
SAM (example)

a
c(w,$\mathcal{E}_w$,)
::
OK

```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

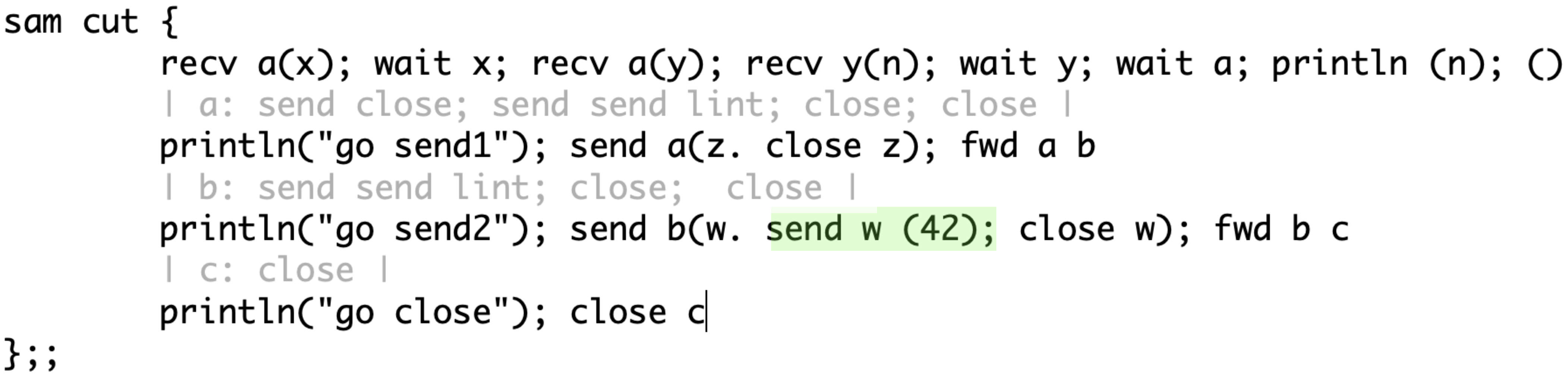
r	a
nil	OK



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

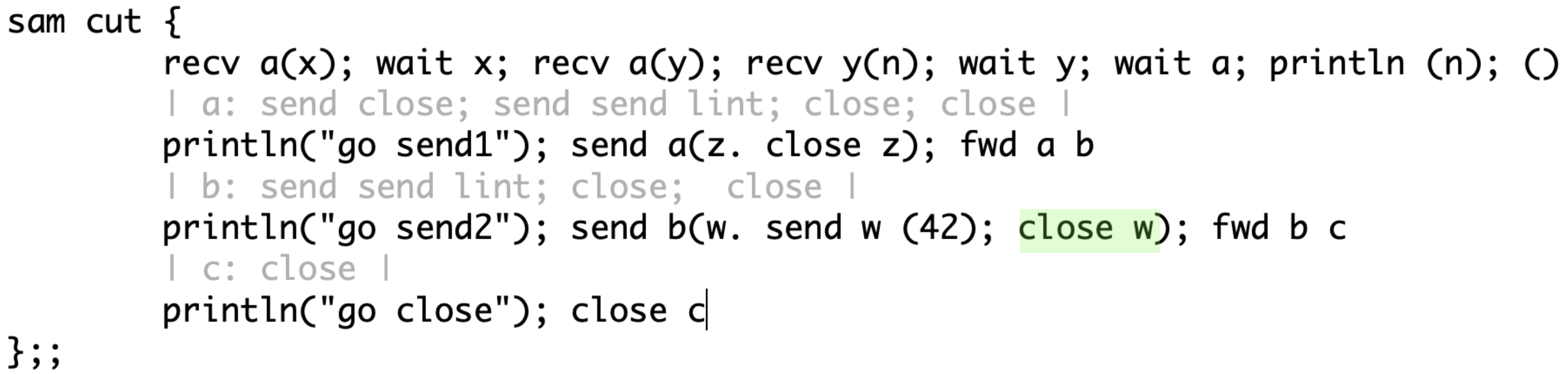
r	a
42	OK



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

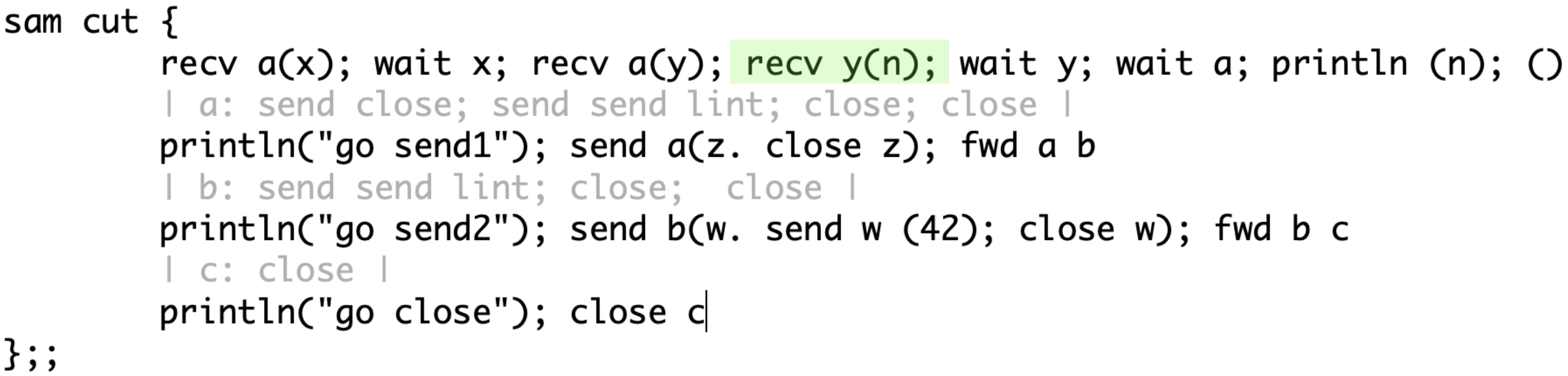
r	a
42 :: OK	OK



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)

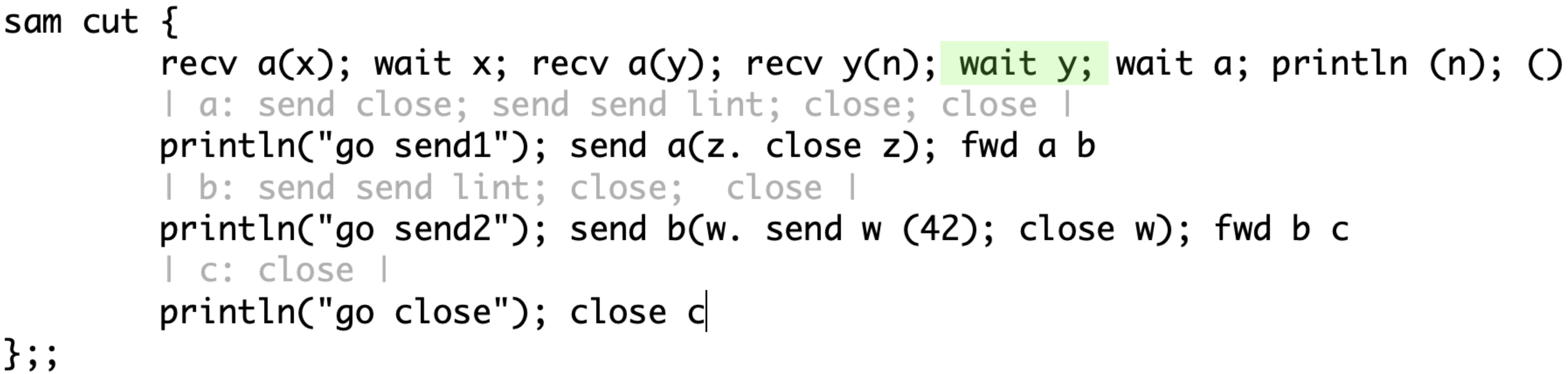
r	a
OK	OK



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

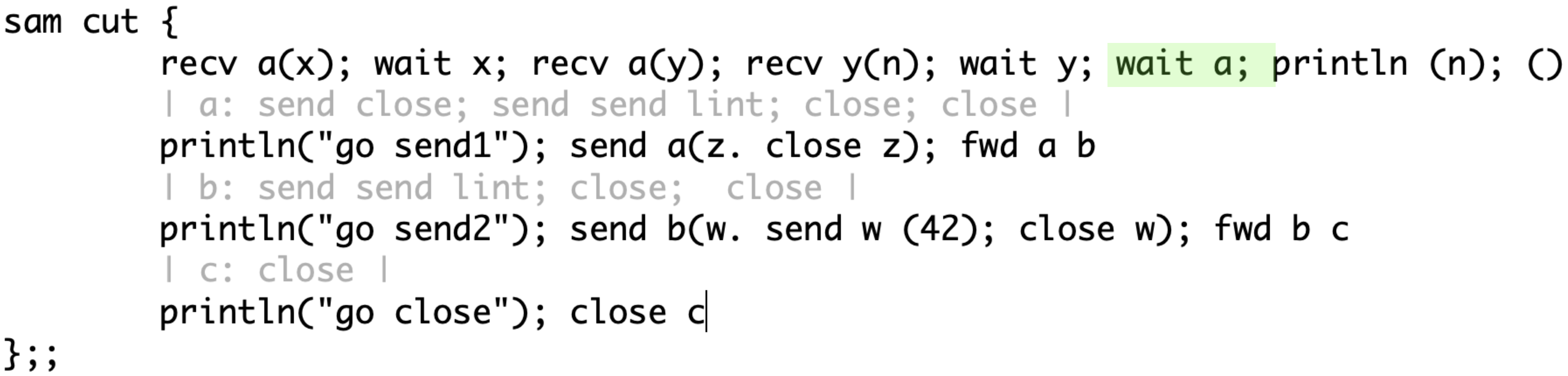
SAM (example)

```
a
OK
```



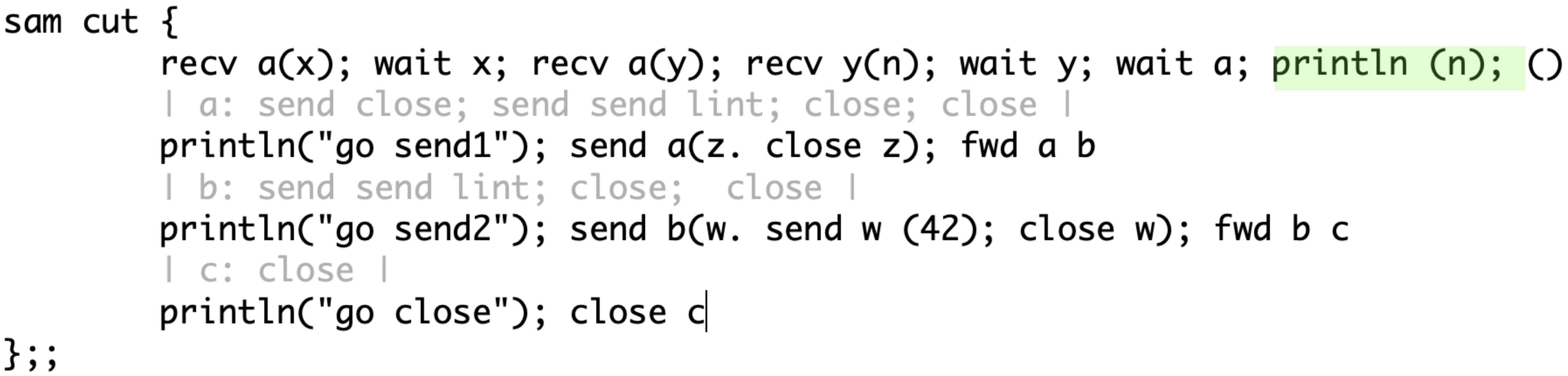
```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

SAM (example)



```
sam cut {
        recv a(x); wait x; recv a(y); recv y(n); wait y; wait a; println (n); ()
        | a: send close; send send lint; close; close |
        println("go send1"); send a(z. close z); fwd a b
        | b: send send lint; close;  close |
        println("go send2"); send b(w. send w (42); close w); fwd b c
        | c: close |
        println("go close"); close c
};;
```

Concluding Remarks

- SAM, a “simple” abstract machine for linear session-based computation
  - factors-out sequential from concurrent computation on linear session calculi.
  - explicit control of memory allocation / deallocation.
- Well-typed programs respect the algebraic operational semantics
- Some “easy” optimizations
  - Queues and environments can be replaced by array-based stack frames.
  - We are on the way of implementing a compiler for CLASS targeting the LLVM.
- We expect SAM to promote adoption of safe session-based concurrent programming,